UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Petar Magdevski

Algoritmi za problem izomorfizma dreves

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Jurij Mihelič

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Algoritmi za problem izomorfizma dreves

Tematika naloge:

Ugotavljanje enakosti podatkovnih struktur je eden izmed ključnih računskih problemov in se pogosto uporablja pri iskanju vzorcev.
V diplomski nalogi se osredotočite na problem ugotavljanja enakosti oz. izomorfnosti dreves. Podajte pregled področja, definirajte osnovne pojme in predstavite različne algoritme za problem izomorfizma dreves. Eksperimentalno preizkusite in primerjajte izbrane algoritme na ustrezno generirani testni množici dreves.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Petar Magdevski, z vpisno številko 63120359, avtor zaključnega dela z naslovom:

*Algoritmi za problem izomorfizma dreves (*angl. *Algorithms for the tree isomorphism problem)*

IZJAVLJAM

1. da sem pisno zaključno delo študija izdelal samostojno pod mentorstvom doc. dr. Jurija Miheliča;
2. da je tiskana oblika pisnega zaključnega dela študija istovetna elektronski obliki pisnega zaključnega dela študija;
3. da sem pridobil vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v pisnem zaključnem delu študija in jih v pisnem zaključnem delu študija jasno označil;
4. da sem pri pripravi pisnega zaključnega dela študija ravnal v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil soglasje etične komisije;
5. soglašam, da se elektronska oblika pisnega zaključnega dela študija uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;
6. da na UL neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja pisnega zaključnega dela študija na voljo javnosti na svetovnem spletu preko Repozitorija UL;
7. dovoljujem objavo svojih osebnih podatkov, ki so navedeni v pisnem zaključnem delu študija in tej izjavi, skupaj z objavo pisnega zaključnega dela študija.

V Ljubljani, dne 1. marca 2016 Podpis študenta:

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
AHU	Aho, Hopcroft and Ullman	Aho, Hopcroft in Ullman

Povzetek

Naslov: Algoritmi za problem izomorfizma dreves

V diplomskem delu so predstavljeni algoritmi za preverjanje enakovrednosti, oz. izomorfnosti dreves. Definirana so drevesa, opisani so osnovni koncepti drevesnih obhodov, podane so podrobnosti algoritmov za drevesni izomorfizem, med katerimi sta bolj podrobno predstavljena algoritma za izomorfizem neurejenih dreves po Valiente ter po Aho, Hopcroft in Ullman. Algoritmi so tudi implementirani ter eksperimentalno primerjani. Primerjava je bila narejena na podlagi časa, ki ga je vsak od algoritmov porabil pri določanju izomorfizma naključno generiranih dreves. Drevesa so bila razvrščena v skupine, odvisno od števila vozlišč in nad vsako skupino dreves sta bila izvedena oba algoritma.

Ključne besede: drevo, izomorfizem, algoritem.

Abstract

Title: Algorithms for the tree isomorphism problem

In this thesis algorithms for checking isomorphism between trees are elaborated. Tree definition is provided, basic concepts of tree traversals are described, details of tree isomorphism algorithms are given, putting special attention and focus on unordered tree isomorphism, algorithm by Valiente as well as algorithm by Aho, Hopcroft and Ullman. Algorithms have been implemented and compared. Comparison was done upon the times needed for each of the algorithms to determine the isomorphism between randomly generated trees. Trees were classified in groups depending on the number of vertices and both algorithms were executed over each group of vertices.

Keywords: tree, isomorphism, algorithm.

Poglavje 1 Uvod

Drevesna identifikacija je problem ugotavljanja, ali sta dve drevesi izomorfni, kar dejansko predstavlja temeljni problem različnih aplikacij v različnih znanstvenih in tehničnih disciplinah.

Glede na to, da so drevesa lahko urejena (angl. ordered) ali neurejena (angl. unordered), obstajajo različni pogledi na drevesni izomorfizem.

S problemom določanja izomorfnosti med drevesi se danes srečujejo in ga tudi uspešno uporabljajo v molekularni biologiji, kjer molekule RNK (ribonukleinska kislina) primerjajo in jih tretirajo kot neusmerjena, povezana ter označena drevesa. Drugo področje, kjer se drevesni izomorfizem uporablja, je primerjanje strukturiranih dokumentov, kot so na primer SGML in XML. Tretje pa detekcija podobnih razredov v programskih jezikih, denimo v Javi [3].

1.1 Motivacija

Osnovni cilj te diplomske naloge je primerjati algoritme za drevesni izomorfizem. Da bi to dosegli, bi jih morali implementirati, uporabiti neko testno okolje, kjer bomo imeli zadostno število dreves z določenim številom vozlišč in izvedli dejansko primerjavo časov med izvajanjem različnih algoritmov.

Opisani algoritmi delujejo le na ukoreninjenih drevesih.

1.2 Pregled dela

V poglavju 2 podamo osnovne definicije dreves ter razložimo vrste in načine obhoda dreves. Opišemo tudi drevesni izomorfizem.

V poglavju 3 razložimo algoritme za drevesni izomorfizem, s poudarkom na dveh algoritmih za neurejena drevesa: po Valiente ter po Aho, Hopcroft in Ullman.

V poglavju 4 smo predstavili eksperimentalno primerjavo algoritmov, kjer smo podrobno razložili način generiranja naključnih dreves, detajle o uporabljenem testnem okolju in dobljene primerjalne rezultate.

V poglavju 5 smo podali zaključek dela ter sklepne ugotovitve.

Poglavje 2 Osnovne definicije

2.1 Drevo

Naj bo V končna neprazna množica in naj bo E poljubna družina dvoelementnih podmnožic množice V. Potem paru $G=(V,E)$ pravimo neusmerjen graf na množici vozlišč $V=V(G)$ in z množico povezav $E=E(G)$. Element $\{u,v\}$ množice E pišemo krajše kot uv.

Zaporedje vozlišč in povezav $S=v_0e_1v_1e_2v_2...v_{k-1}e_kv_k$, kjer je $e_i=v_{i-1}v_i$ imenujemo sprehod v grafu. Sprehod je enostaven, kadar so vse povezave $e_1,e_2,\ldots,e_k$ med seboj različne. Enostaven sprehod je cikel, kadar so vse točke $v_0,v_1,\ldots v_{k-1}$ med seboj različne in $v_0=v_k$ [5].

Drevo (angl. tree) $T = (V, E)$ predstavlja neusmerjen povezan graf s končno množico vozlišč V (angl. vertices) in končno množico povezav E (angl. edges), v katerem ni ciklov.

Drevo s korenom ali ukoreninjeno drevo $T = (V, E, r)$ sestoji iz končne množice vozlišč V, končne množice povezav E. Obstaja eno določeno vozlišče $r \in V$, ki je koren drevesa (angl. root) in ga označimo z root[T] [6].

Pri drevesu s korenom poznamo relacijo starš-otrok oziroma prednik-potomec. Na sliki 2.1 v drevesu T2 je vozlišče A starš vozlišč B in C oziroma vozlišči B in C sta otroka vozlišča A. Vozlišče C je prednik vozlišč G, H in I oziroma vozlišča G, H in I so potomci vozlišča C. Vrstni red otrok štejemo z leve proti desni. Prvi otrok (angl. first child) je vsako prvo vozlišče med otroci določenega vozlišča, medtem ko je zadnji otrok zadnje vozlišče med otroci določenega vozlišča. List (angl. leaf) je vozlišče v drevesu, ki nima otrok in je povezano le s svojim staršem.

Samo eno vozlišče v nepraznem drevesu je brez starša in mu pravimo koren.

Vsako vozlišče ima lahko nič ali več otrok.

Nivo vozlišča je dolžina poti od korena do vozlišča [7]. Globina drevesa je dolžina najdaljše poti od korena do lista drevesa.

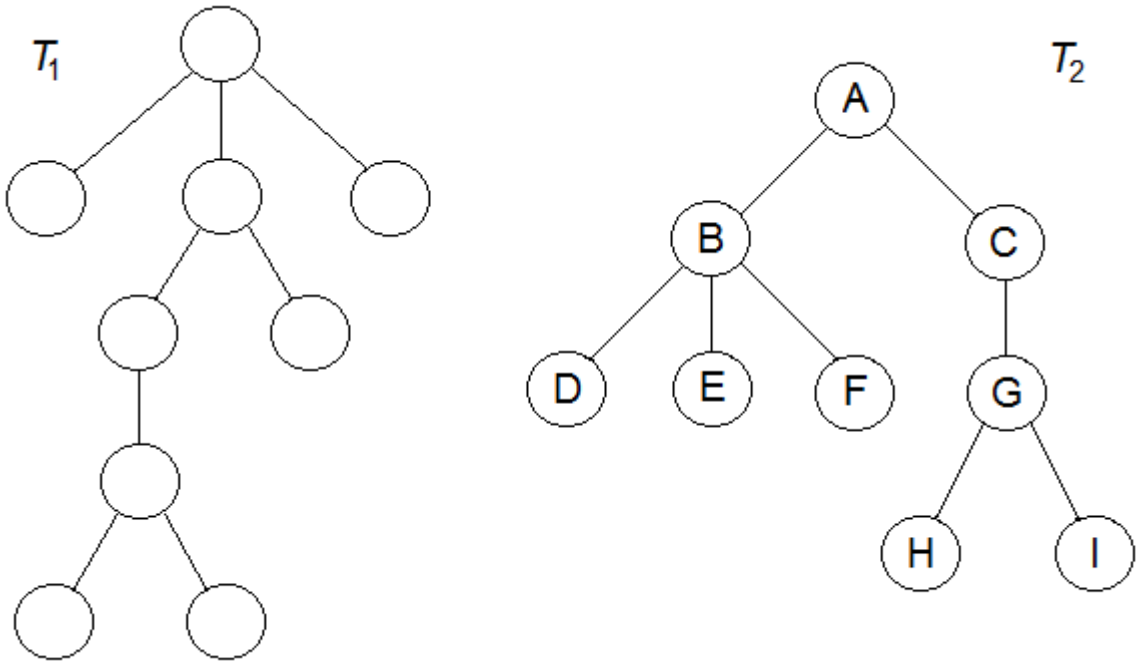
Poddrevo sestavljajo vozlišče, ki bo koren poddrevesa in vsi njegovi potomci. Poddrevesa so prav tako drevesa.

Potomci korena razpadejo na disjunktne množice poddreves.

Na sliki 2.1 je drevo T_1 neoznačeno drevo (angl. unlabeled tree), T_2 pa označeno drevo (angl. labeled tree). Oznake označenega drevesa so v vozliščih. Koren drevesa T_2 je vozlišče A.

Glede na urejenost med starši in otroki ter vrstni red otrok, poznamo urejena (angl. ordered trees), kjer so vozlišča urejena po določenih kriterijih in neurejena drevesa (angl. unordered trees), kjer vozlišča niso urejena.

Gozd (angl. forest) je pa množica dreves.



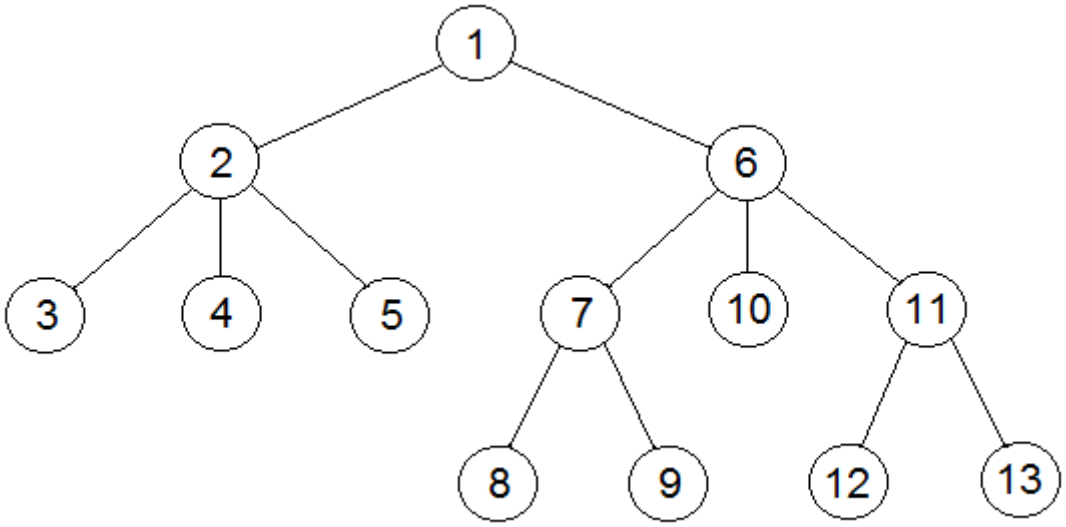
Slika 2.1: Drevesa: T_1 - neoznačeno drevo, T_2 - označeno drevo.

2.2 Drevesni obhodi

Večina drevesnih algoritmov zahteva nek sistematičen način obiskovanja vseh vozlišč drevesa. Obhod drevesa $T = (V, E)$ z n vozlišči je bijekcija: $V \rightarrow \{1, ..., n\}$. Obhod skozi drevo se začne z obiskom prvega vozlišča u, ki ima red 1 (order[u]=1), nadaljuje se z vozliščem v z redom 2 (order[v]=2) in tako naprej, dokler se kot zadnje n-to ne obišče vozlišče z (order[z]=n).

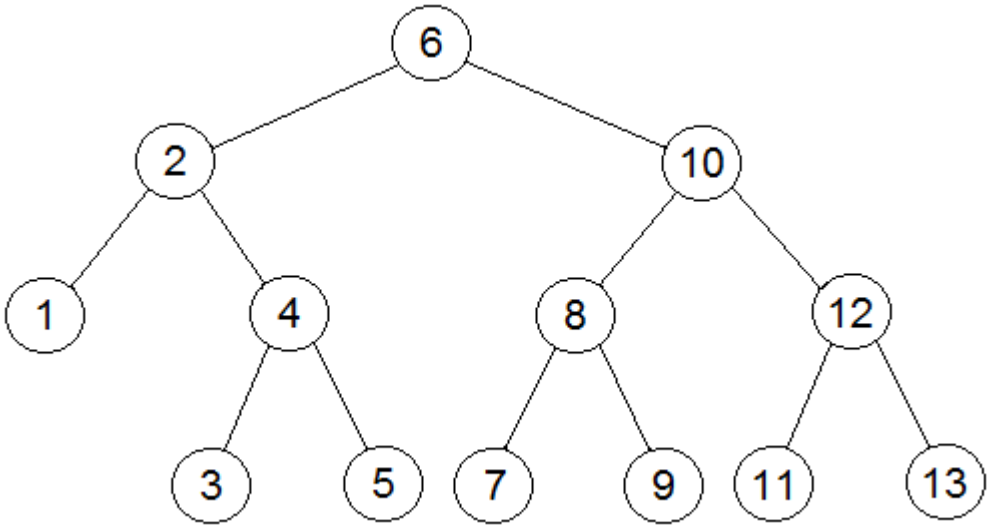
Najpogostejši načini obhodov dreves so premi obhod (angl. preorder), vmesni obhod (angl. inorder), ki velja le za dvojiška drevesa, obratni obhod (angl. postorder) in obhod po nivojih (angl. level order) od zgoraj navzdol (angl. top-down).

Pri premem obhodu drevesa s korenom, je starš obiskan pred otroci, medtem ko so vozlišča, ki se nahajajo na istem nivoju (otroci) obiskana po vrsti z leve proti desni. Na sliki 2.2 je prikazan premi obhod drevesa s korenom.



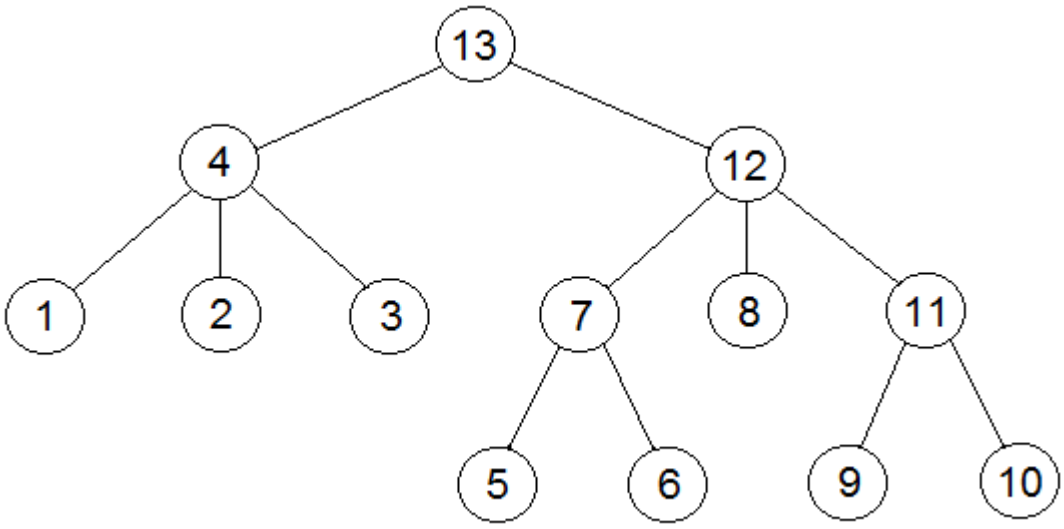
Slika 2.2: Premi obhod drevesa s korenom. Vozlišča so oštevilčena po vrstnem redu obiska med obhodom.

Pri vmesnem obhodu, prikazanem na sliki 2.3 se najprej obišče levi otrok (levo poddrevo), nato njegov starš (ki pravzaprav predstavlja njegov koren) in šele potem desni otrok (desno poddrevo).



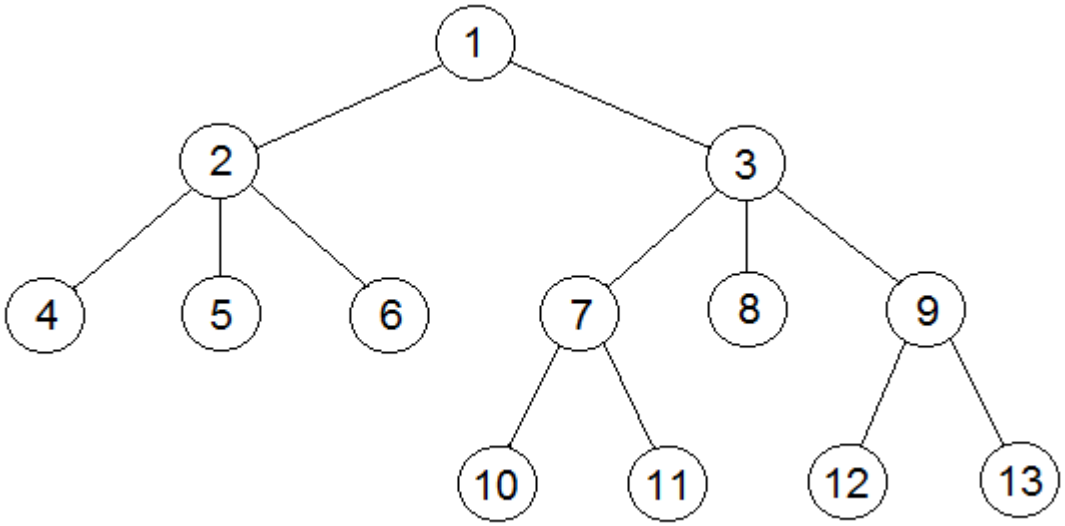
Slika 2.3: Vmesni obhod dvojiškega drevesa. Vozlišča so oštevilčena po vrstnem redu obiska med obhodom.

Pri obratnem obhodu drevesa s korenom, prikazanem na sliki 2.4 so otroci obiskani pred starši. Vozlišča na istem nivoju so obiskana z leve proti desni.



Slika 2.4: Obratni obhod drevesa s slike 2.2. Vozlišča so oštevilčena po vrstnem redu obiska med obhodom.

Pri obhodu ukoreninjenega drevesa od zgoraj navzdol, znanem tudi kot obhod po nivojih, so vozlišča obiskana po nepadajoči globini. Ta obhod je prikazan na sliki 2.5.



Slika 2.5: Obhod drevesa s korenom s slike 2.2 z obhodom od zgoraj navzdol. Vozlišča so oštevilčena po vrstnem redu obiska med obhodom.

2.3 Izomorfizem

Naj sta podani drevesi $T_1 = (V_1, E_1)$ in $T_2 = (V_2, E_2)$. Izomorfizem dreves T_1 in T_2 je bijektivna preslikava $M : V_1 \rightarrow V_2$, za katero velja $(u, v) \in E_1 \Leftrightarrow (M(u), M(v)) \in E_2$, pri čemer $u \in V_1$ in $v \in V_2$.

Drevesi $T_1 = (V_1, E_1)$ in $T_2 = (V_2, E_2)$ sta izomorfni, kadar med njima obstaja izomorfizem. Izomorfizem ohranja strukturo ter povezanost drevesa. To zapišemo z $T_1 \cong T_2$.

Problem izomorfizma dreves je za dve dani drevesi ugotoviti, ali sta izomorfni. To je eden osnovnih problemov z različnimi izvedbenimi rešitvami.

V naslednjem poglavju so podani primeri ter podrobnejša obrazložitev izomorfizma dreves.

Poglavje 3 Algoritmi

3.1 Izomorfizem urejenih dreves

Definicija 1. Dve urejeni drevesi (angl. ordered trees) s korenom $T_1 = (V_1, E_1)$ in $T_2 = (V_2, E_2)$ sta izomorfni $T_1 \cong T_2$, če obstaja bijekcija $M : V_1 \rightarrow V_2$, tako da je $M(\text{root}[T_1]) = \text{root}[T_2]$ in veljajo naslednje trditve:

- $M(\text{first}[u]) = \text{first}[v]$, za vsa vozlišča $u \in V_1$ in vsa vozlišča $v \in V_2$, ki niso listi ter $M(u)=v$
- $M(\text{next}[u]) = \text{next}[v]$, za vsa vozlišča $u \in V_1$ in vsa vozlišča $v \in V_2$, ki niso zadnji otroci ter $M(u)=v$

kjer root[] predstavlja koren drevesa, first[] je prvo vozlišče med otroci določenega vozlišča, next[] pa naslednje, medtem ko zadnji otrok predstavlja zadnje vozlišče med otroci določenega vozlišča. M je urejen drevesni izomorfizem iz T_1 v T_2 [1].

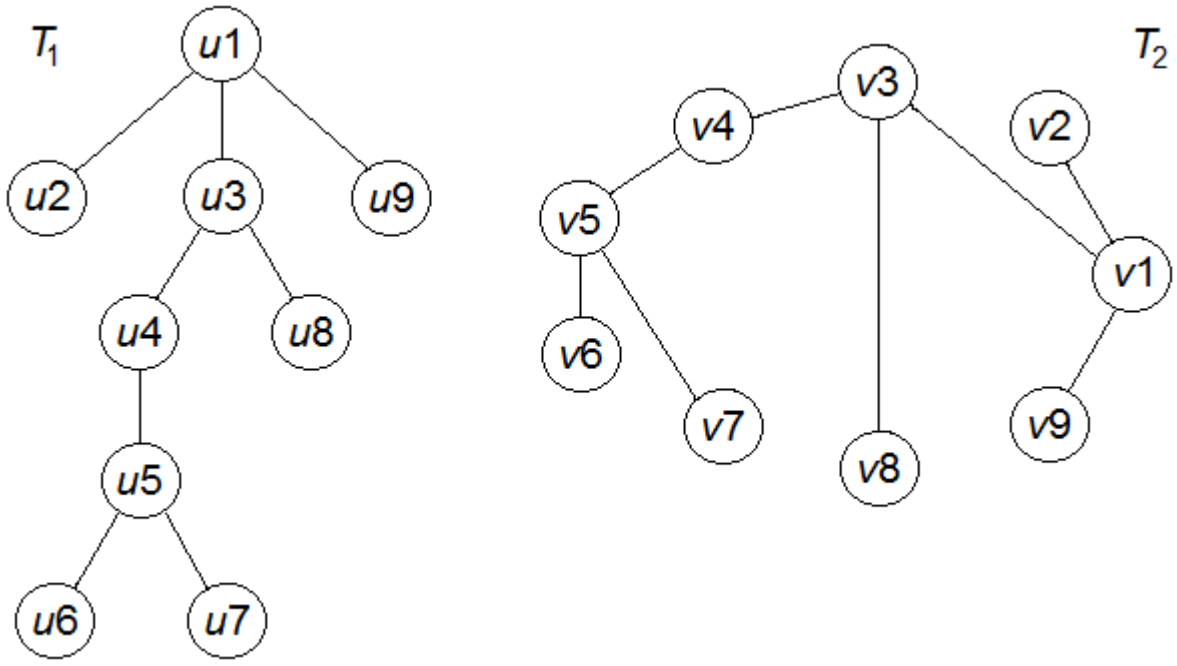
Dve ukoreninjeni urejeni drevesi sta izomorfni, če obstaja bijektivna preslikava M med množicami njunih vozlišč V_1 in V_2, ki ohranja strukturo obeh dreves. Vozlišče, ki ustreza korenu prvega drevesa je koren drugega drevesa $M(\text{root}[T_1]) = \text{root}[T_2]$, vozlišče v_1 je starš vozlišča v_2, če in samo če je vozlišče, ki ustreza vozlišču v_1 starš vozlišča, ki ustreza vozlišču v_2 v drugem drevesu, ter vozlišče v_3 je naslednji sorojenec vozlišča v_2, če je vozlišče, ki ustreza vozlišču v_3 prav tako naslednji sorojenec vozlišču, ki ustreza vozlišču v_2 v drugem drevesu.

3.1.1 Drevesni obhod

Naslednji postopek, ki se uporablja pri algoritmu po Valiente za urejena drevesa, izvaja obhod drevesa s korenom $T = (V, E)$ od zgoraj navzdol z uporabo vrste (angl. queue) vozlišč Q. Vrstni red, v katerem so vozlišča obiskana med obhodom od zgoraj navzdol, bo shranjen v polju vozlišč *order*.

```
void top_down_tree_traversal(tree T, nodearray<int> order) {
  queue<node> Q;
  node u, v;
  Q.append(T.root());
  int num=1;
  do {
    u=Q.pop();
    order[u]=num++;
    forallchildren(v, u) {
      Q.append(v);
    }
  } while (!Q.empty());
}
```

3.1.2 Algoritem po Valiente za urejena drevesa



Slika 3.1: Izomorfna urejena drevesa. Vozlišča so oštevilčena po vrstnem redu obiska med premim obhodom.

Urejeni drevesi $T_1 = (V_1, E_1)$ in $T_2 = (V_2, E_2)$ s slike 3.1, sta izomorfni. Bijekcija $M : V_1 \rightarrow V_2$, tako da $M = \{(u_1, v_1), (u_2, v_2), (u_3, v_3), (u_4, v_4), (u_5, v_5), (u_6, v_6), (u_7, v_7), (u_8, v_8), (u_9, v_9)\}$ predstavlja urejen drevesni izomorfizem iz T_1 v T_2.

Najbolj enostaven postopek za določanje izomorfnosti dveh urejenih dreves bi bil obhod skozi obe drevesi z enakim obhodom, recimo z obhodom od zgoraj navzdol, na kar bi sledilo

preverjanje, ali je preslikava vozlišč, ki smo jo dobili z obhodom, izomorfizem med urejenimi drevesi [1]. Za vozlišče $u \in V$ v drevesu $T = (V, E)$, $T[u]$ je oznaka vozlišča u.

```
bool simple_ordered_tree_isomorphic (tree T1, tree T2, nodearray<node> M) {
  int n=T2.numberofnodes();
  if (T1.numberofnodes()≠n) return false;

  nodearray<int> order1(T1);
  nodearray<int> order2(T2);
  top_down_tree_traversal(T1, order1);
  top_down_tree_traversal(T2, order2);

  array<node> disorder2(1, n);
  node u, v;
  forallnodes(u, T2) {
    disorder2[order2[u]]=u;
  }
  forallnodes(u, T1) {
    M[u]=disorder2[order1[u]];
  }
  forallnodes(u, T1) {
    v=M[u];
    if (T1[u]≠T2[v]) return false;
    if          (!T1.isleaf(u)          AND          (T2.isleaf(v)          OR
(M[T1.firstchild(u)]≠T2.firstchild(v)))) return false;
    if       (!T1.islastchild(u)       AND       (T2.islastchild(v)       OR
(M[T1.nextsibling(u)]≠T2.nextsibling(v)))) return false;
  }
  return true;
}
```

3.1.3 Izboljšana različica algoritma po Valiente za urejena drevesa

Algoritem za določanje izomorfnosti urejenih dreves se lahko izboljša na ta način, da se preverjanje neizomorfnosti dveh urejenih dreves naredi prej, z istočasnim obhodom skozi obe drevesi namesto da se izvaja obhod vsakega drevesa ter preizkušanje inducirane preslikave vozlišč.

```
bool ordered_tree_isomorphism(tree T1, tree T2, nodearray<node> M) {
  if (T1.numberofnodes()≠T2.numberofnodes()) return false;
  if (map_ordered_tree(T1, T1.root(), T2, T2.root(), M)) {
    double_check_ordered_tree_isomorphism();
    return true;
  }
  return false;
}
```

```
bool map_ordered_tree(tree T1, node r1, tree T2, node r2, nodearray<node>
M) {
  if (T1[r1]≠T2[r2]) return false;
  M[r1]=r2;
  int d1=T1.numberofchildren(r1);
  int d2=T2.numberofchildren(r2);
  if (d1≠d2) return false;
  node v1, v2;
  if (!T1.isleaf(r1)) {
    v1=T1.firstchild(r1);
    v2=T2.firstchild(r2);
    if (!map_ordered_tree(T1, v1, T2, v2, M)) return false;
    for (int i=2; i≤d1; i++) {
      v1=T1.nextsibling(v1);
      v2=T2.nextsibling(v2);
      if (!map_ordered_tree(T1, v1, T2, v2, M)) return false;
    }
  }
  return true;
}
```

Naslednje preverjanje urejenega drevesnega izomorfizma, čeprav je odveč, zagotavlja pravilnost izvajanja postopka ter potrjuje, da je *M* urejen drevesni izomorfizem iz T_1 v T_2 po definiciji 1.

```
void double_check_ordered_tree_isomorphism() {
  node v;
  forallnodes(v,T1) {
    if (M[v]≡null OR T1[v]≠T2[M[v]])
      error_handler("Wrong implementation of ordered tree isomorphism");
    if         (!T1.isleaf(v)         AND         (T2.isleaf(M[v])         OR
(M[T1.firstchild(v)]≠T2.firstchild(M[v]))))
      error_handler("Wrong implementation of ordered tree isomorphism");
    if      (!T1.is_lastchild(v)      AND      (T2.islastchild(M[v])      OR
(M[T1.nextsibling(v)]≠T2.nextsibling(M[v]))))
      error_handler ("Wrong implementation of ordered tree isomorphism");
  }
}
```

Glede na to, da ta algoritem ugotavlja izomorfnost dreves v primeru ko imajo le-ta enako strukturo, nam algoritem za preverjanje enakosti urejenih dreves ni v interesu. Fokusirali se bomo na naslednja dva algoritma za določanje izomorfizma neurejenih dreves.

3.2 Leksikografsko urejanje besedil

Urejanje je praktično pomemben, teoretično zanimiv problem. Postopek razvrščanja zaporedja elementov je pomemben del številnih algoritmov. Pogledali si bomo le nekatere algoritme za urejanje.

3.2.1 Urejanje s koši

Naj bo a_1, a_2, ..., a_n zaporedje celih števil od 0 do m-1. Če m ni preveliko število lahko za urejanje zaporedja uporabimo naslednji postopek [2]:

1. Inicializacija m praznih vrst (angl. queue), po eno za vsako število v območju 0 do m-1. Vsaka vrsta se imenuje koš (angl. bucket)

2. Pregledovanje zaporedja a_1, a_2, ..., a_n od leve proti desni, z dodajanjem elementa a_i v a_i-to vrsto

3. Spojitev vrst (vsebina vrste i+1 se pripne na konec vrste i) z namenom dobiti urejeno zaporedje

Ker se en element lahko doda v i-to vrsto v konstantnem času, n elementov se lahko vstavi v vrste v času $O(n)$. Spojitev m vrst zahteva $O(m)$ časa. Celotna časovna zahtevnost algoritma je $O(n+m)$. Če je m majhen oziroma $m=O(n)$, potem za urejanje n celih števil algoritem ima časovno zahtevnost $O(n)$. Algoritem, ki razdeli elemente zaporedja v določeno število košev in vsak koš naknadno uredi je znan kot urejanje s koši (angl. bucket sort). Algoritem se lahko razširi in uporabi za urejanje zaporedja terk (angl. tuples) celih števil v leksikografskem vrstnem redu. Na primer, če se nizi črk, podani v abecednem vrstnem redu obravnavajo kot terke, potem so besede v slovarju podane v leksikografskem vrstnem redu.

3.2.2 Korensko urejanje

Eden od načinov urejanja celih števil v razponu od 0 do m-1 je korensko urejanje (angl. radix sort), ki ureja števila po števkah od najmanj pomembne do najbolj pomembne oziroma z desne proti levi (angl. least significant digit radix sort).

Na splošno, urejanje zaporedja k-terk, sestavljenih iz celih številk v razponu od 0 do m-1, dela tako, da naredi k sprehodov nad zaporedjem z uporabo urejanja s koši pri vsakem sprehodu. Pri prvem sprehodu so k-terke urejene po njihovi k-ti komponenti, po drugem sprehodu, bo zaporedje, ki je nastalo iz prvega sprehoda, urejeno po (k-1) komponenti, po tretjem, bo nastalo zaporedje urejeno glede po (k-2) komponenti, itd. Po k-em in zadnjem sprehodu, bo

zaporedje, ki je nastalo iz (k-1) sprehoda, urejeno po prvih komponentah in je v leksikografskem vrstnem redu. Podrobnejši opis sledi v nadaljevanju.

Pri leksikografskem urejanju se iz zaporedja A_1, A_2, ..., A_n, kjer vsaka A_i je k-terka (a_{i1}, a_{i2}, ..., a_{ik}) in a_{ij} je celo število v razponu od 0 do m-1, dobi zaporedje B_1, B_2, ..., B_n, ki je pravzaprav permutacija zaporedja A_1, A_2, ..., A_n, tako da $B_i \leq B_{i+1}$ za $1 \leq i < n$.

Vrsta, imenovana *QUEUE*, se uporablja za hranjenje trenutnega zaporedja elementov. Prav tako se uporablja niz Q m košev, pri čemer je koš *Q[i]* namenjeno tistim k-terk, ki imajo številko i v komponenti, ki se trenutno obravnava:

```
void lexicographic_sort() {
  queue QUEUE;
  array Q;

  for (int i=1; i≤n; i++) {
    QUEUE.append(A(i)); //place A1, A2, ..., An in QUEUE
  }
  for (int j=k; j≥1; j--) {
    for (int l=0; l≤m-1; l++) {
      Q[l].clear; //make Q[l] empty
    }
    while (!QUEUE.empty()) {
     x=QUEUE.pop(); //x=Ai be the first element in QUEUE
      Q[x(j)]=x;   //move x from QUEUE to bucket Q[x(j)]
    }
    for (int l=0; l≤m-1; l++) {
      //concatenate contents of Q[l] to the end of QUEUE
      QUEUE.append(Q[l]);
    }
  }
}
```

Ta algoritem ima različne uporabe. Druga posplošitev algoritma za urejanje s koši bi bila na zaporedjih različnih dolžin, ki jih bomo poimenovali nizi. V primeru, da ima najdaljši niz dolžino k, bi lahko vsakemu od krajših nizov dodali na koncu en določen specialen simbol tolikokrat, da imajo vsi nizi dolžino k in šele potem bi lahko uporabili prejšnji algoritem. Vendar, če obstaja le peščica dolgih nizov, bi bil ta pristop neučinkovit iz dveh razlogov: prvič, vsak niz se obdela na vsakem sprehodu in drugič, vsak koš *Q[i]* je obdelano tudi v primeru, če so vsi koši skoraj prazni.

V nadaljevanju bomo opisali algoritem, ki ureja zaporedje *n* nizov različnih dolžin, v razponu od 0 do *m*-1, s časovno zahtevnostjo *O(m+*l_{total}*)*, kjer je l_i dolžina *i*-ga niza in $l_{total} = \sum_{i=1}^{n} l_i$. Algoritem je uporaben, ko imata *m* in l_{total} zahtevnost *O(n)*.

Bistvo algoritma je, da najprej razvrsti nize v padajoči vrstni red. Naj je l_{max} dolžina najdaljšega niza. Urejanje s koši oziroma sprehodi se ponovijo l_{max}-krat po zgoraj opisanem algoritmu. V prvem sprehodu se uredijo po skrajno desni komponenti le nizi z dolžino l_{max}. V drugem sprehodu se po ključu l_{max}-1 komponente uredijo nizi, ki imajo dolžino vsaj l_{max}-1, itd.

Omenili bi še to, da komponente terk niso nujno le števila, so lahko tudi črke. Za ilustracijo si oglejmo primer, kako bi uredili tri nize *bab*, *abc* in *a*. *a*, *b* in *c* se lahko zamenjajo z številkami 0,1 in 2. V tem primeru je l_{max}=3, tako da se v prvem sprehodu na podlagi tretje komponente urejata le prva dva niza. Niz *bab* bo šel v *b*-koš in niz *abc* v *c*-koš, medtem ko bo *a*-koš ostal prazen. Na podlagi druge komponente v drugem sprehodu bosta urejena spet ista niza; *a*-koš in *b*-koš bosta zasedena (zaradi *bab* in *abc*) in *c*-koš bo ostal prazen. V tretjem in zadnjem sprehodu, bodo vsi trije nizi urejeni na podlagi prve komponente, tako da bosta *a*-koš in *b*-koš zasadena, medtem ko bo *c*-koš prazen.

Opažamo, da pri sprehodih obstajajo prazni koši. Dodatni korak, s katerim bi se ugotovilo, kateri koši bodo zasedeni, bi bil koristen. Seznam zasedenih košev za vsak sprehod je določen v naraščajočem vrstnem redu glede na številko koša. To omogoča spojitev zasedenih košev v času sorazmernem številu zasedenih košev.

3.2.3 Izboljšano urejanje

Naj bodo A_1, A_2, ..., A_n terke (zaporedja nizov) različnih dolžin celih števil v razponu od 0 do *m*-1, l_i pa dolžine terke A_i = (a_{i1}, a_{i2}, ..., a_{il_i}) in l_{max} največji l_i. Leksikografsko urejanje terke A_1, A_2, ..., A_n je njegova permutacija B_1, B_2, ..., B_n tako, da je $B_1 \leq B_2 \leq \ldots \leq B_n$ in poteka v treh korakih na naslednji način:

1. Ustvariti sezname, po enega za vsak l, $1 \leq l \leq l_{max}$, za simbole, ki se pojavljajo v *i*-ti komponenti enega ali večih nizov. Najprej se ustvarijo pari (l, a_{il}) za vsako komponento a_{il}, $1 \leq i \leq n$, $1 \leq l \leq l_i$ in vsakega niza A_i, po en par. Takšen par označuje, da *l*-ta komponenta nekega niza vsebuje celo število a_{il}. Nato se ti pari leksikografsko uredijo s prejšnjem algoritmom, pri čemer je prva komponenta v razponu od 1 do l_{max} in druga od 0 do *m*-1. Potem se s pregledovanjem tako urejenega seznama od leve proti desni lahko naredijo l_{max} urejeni seznami NONEMPTY[l], $1 \leq l \leq l_{max}$, takšni da NONEMPTY[l] vsebuje le simbole, ki se

pojavljajo v l-ti komponenti nekaterih nizov. To pomeni, da NONEMPTY[l] vsebuje urejene vse številke j, za katere velja $a_{il} = j$, za določen i.

2. Določiti dolžino vsakega niza. Sledi ustvarjanje seznamov LENGTH[l], za $1 \leq l \leq l_{max}$, kjer LENGTH[l] vsebuje vse nize z dolžino l.

3. Urediti nize po komponentah kot v prejšnjem algoritmu, začenši s komponentami, ki so na položaju l. Kakorkoli, po i-tem sprehodu bo QUEUE vseboval le tiste nize, ki imajo dolžino vsaj l_{max}-i+1, in ti nizi bodo že urejeni po komponentah od l_{max}-i+1 do l_{max}. Seznami NONEMPTY, izračunani v prvem koraku se bodo uporabili pri določanju košev, ki so dejansko zasedena pri vsakem sprehodu urejanja s koši. In prav to se uporablja za pospešitev spojitev košev.

```
void lexicographic_sort_of_strings_of_varying_length() {
  queue QUEUE;
  array Q;

  QUEUE.clear(); //make QUEUE empty
  for (int j=0; j≤m-1; j++) Q[j].clear();
  for (int l=l_max; j≥1; j--) {
    //concatenate LENGTH[l] to the beggining of QUEUE
    QUEUE.add(LENGTH[l]);
    while (!QUEUE.empty()) {
      A(i)=QUEUE.pop();
      Q(a(i,j))=A(i);
    }
    int j;
    foreach(j, NONEMPTY[l]) {
      //concatenate Q[j] to the end of QUEUE
      QUEUE.append(Q[j]);
      Q[j].clear();
    }
  }
}
```

Poglejmo si primer uporabe algoritma za urejanje nizov nad *a*, *bab* in *abc*. V prvem koraku se ustvarijo pari (1, *a*) za prvi niz, (1, *b*), (2, *a*), (3, *b*) za drugi in (1, *a*), (2, *b*), (3, *c*) za zadnji niz. Urejen seznam teh parov je: (1, *a*) (1, *a*) (1, *b*) (2, *a*) (2, *b*) (3, *b*) (3, *c*). S pregledovanjem tega urejenega seznama od leve proti desni, sklepamo, da: NONEMPTY[1] = *a*, *b*, NONEMPTY[2] = *a*, *b*, NONEMPTY[3] = *b*, *c*. V drugem koraku, se izračunajo l_1 = 1, l_2 = 3, l_3 = 3 ter LENGTH[1] = *a*, LENGTH[2] je prazen, LENGTH[3] = *bab*, *abc*. V tretjem in zadnjem koraku algoritma se izračuna QUEUE = *bab*, *abc* in se uredita ta dva niza po njuni

tretji komponenti. Dejstvo, da je NONEMPTY[3] = *b*, *c*, zagotavlja da pri ustvarjanju urejenega seznama, *Q*[*a*] ne bo dodan na konec QUEUE. Torej bo po prvem sprehodu QUEUE = *bab*, *abc*. V drugem sprehodu se QUEUE ne spremeni, ker je LENGTH[2] prazen in urejanje po drugi komponenti ne spremeni vrstnega reda. V tretjem sprehodu, se najprej, zaradi dodajanja LENGTH[1] na začetek QUEUE, postavi QUEUE = *a*, *bab*, *abc*, nato sledi urejanje po prvih komponentah, kar pa da QUEUE = *a*, *abc*, *bab*, ki je pravilen vrstni red. *Q*[*c*] se v tretjem sprehodu ne doda na konec QUEUE, ker je *Q*[*c*] prazen in *c* ne pripada NONEMPTY[1].

3.3 Izomorfizem neurejenih dreves

Definicija 2. Dve neurejeni drevesi (angl. unordered trees) s korenom $T_1 = (V_1, E_1)$ in $T_2 = (V_2, E_2)$ sta izomorfni $T_1 \cong T_2$, če obstaja bijekcija $M : V_1 \rightarrow V_2$, tako da $M(\text{root}[T_1]) = \text{root}[T_2]$ in naslednja trditev je tudi izpolnjena:

- $M(\text{parent}[u]) = \text{parent}[v]$, za vsa vozlišča, ki niso koren, $u \in V_1$ in $v \in V_2$ ter $M(u)=v$

kjer je root[] koren drevesa in parent[] predstavlja starša določenega vozlišča. V tem primeru je *M* neurejen drevesni izomorfizem ali samo drevesni izomorfizem, iz T_1 v T_2 [1].

V manj formalnem jeziku to pomeni, da izomorfizem neurejenih dreves izraža tisto, da sta dve drevesi pravzaprav enaki, če bi permutirali njuni poddrevesi v enem vozlišču, ki predstavlja koren teh poddreves.

Dve ukoreninjeni neurejeni drevesi sta izomorfni, če obstaja bijektivna preslikava med njunimi vozlišči, ki ohranja strukturo obeh dreves, oz. vozlišče, ki ustreza korenu prvega drevesa je koren drugega drevesa in vozlišče v_1 je starš vozlišča v_2, če in samo če je vozlišče, ki ustreza vozlišču v_1 starš vozlišča, ki ustreza vozlišču v_2 v drugem drevesu.

Dve neurejeni označeni drevesi sta izomorfni, če sta osnovni neurejeni drevesi izomorfni in imajo ustrezna vozlišča isto oznako.

Da bi oba algoritma za določanje izomorfizma neurejenih dreves (po Valiente in po Aho, Hopcroft in Ullman), primerjali pod enakimi pogoji, smo namesto korenskega urejanja, na podlagi javanske metode *Java.util.Arrays.sort()* razvili in uporabili lastno metodo za urejanje.

3.3.1 Drevesni obhodi

Naslednji postopek, ki se uporablja pri algoritmu po Valiente za neurejena drevesa, opravlja iterativni obratni obhod skozi neurejeno drevo $T = (V, E)$ s pomočjo sklada (angl. stack) vozlišč S in pri tem gradi seznam vozlišč L v vrstnem redu, v katerem so vozlišča obiskana med obhodom.

```
void postorder_tree_list_traversal(tree T, list<node> L) {
  L.clear();
  stack<node> S;
  S.push(T.root());
  node u, v;
  do {
    u=S.pop();
    L.push(u);
    forallchildren(v, u) {
      S.push(v);
    }
  } while (!S.empty());
}
```

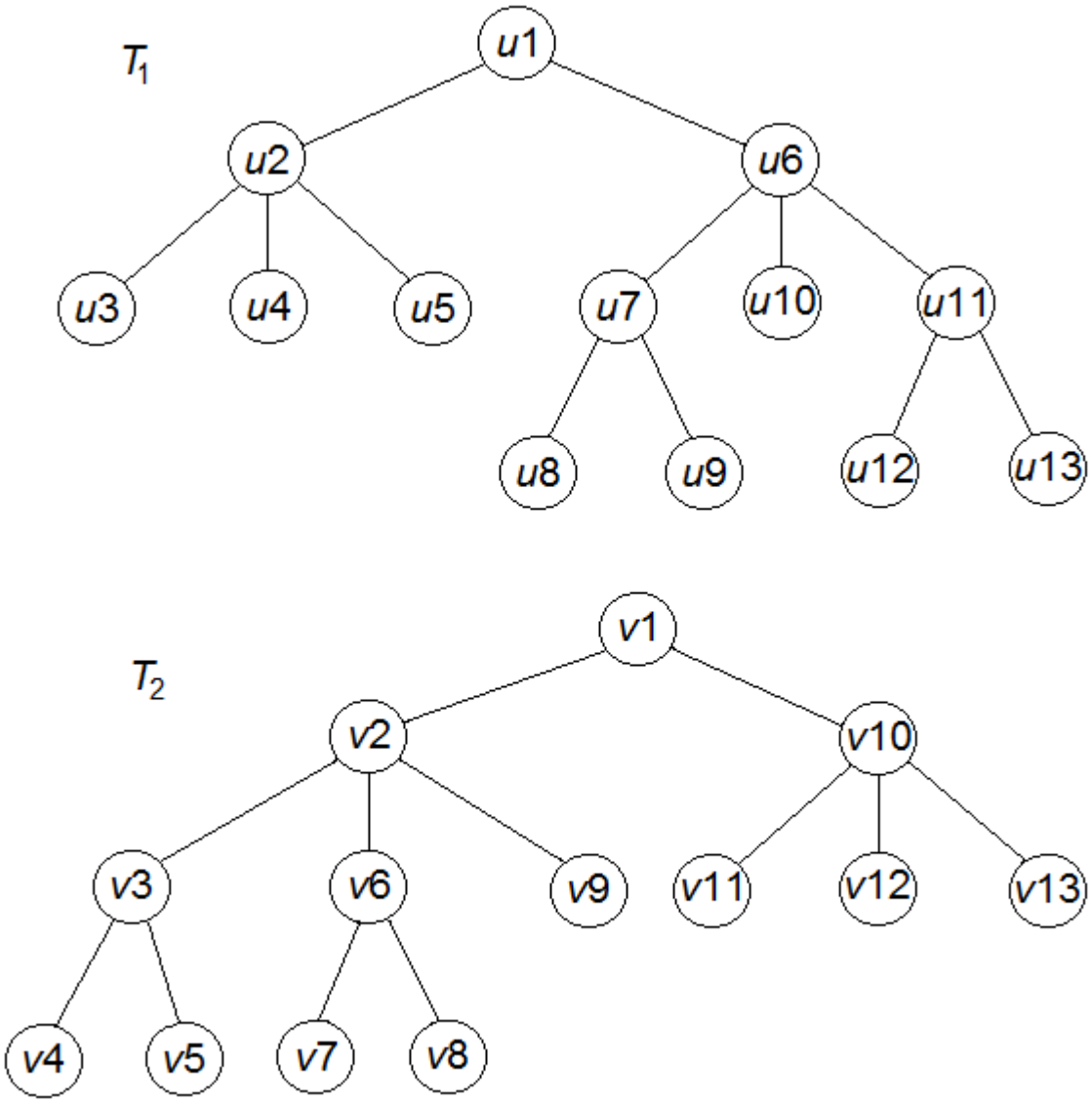
Naslednji algoritem, prav tako uporabljen pri algoritmu po Valiente za neurejena drevesa, izvaja iterativni premi obhod skozi neurejeno drevo T s pomočjo sklada vozlišč S in pri tem gradi seznam vozlišč L v vrstnem redu, v katerem so vozlišča obiskana med obhodom.

```
void preorder_tree_list_traversal(tree T, list<node> L) {
  L.clear();
  stack<node> S;
  S.push(T.root());
  node u, v;
  do {
    u=S.pop();
    L.push(u);
    v=T.last_child(u);
    while (v≠null) {
      S.push(v);
      v = T.previoussibling(v);
    }
  } while (!S.empty());
  L.reverse();
}
```

Postopek, ki opravlja obhod drevesa od zgoraj navzdol in pri tem razvršča vozlišča v ločene skupine, odvisno od globine, na kateri se vozlišča nahajajo, je podan v nadaljevanju. Uporablja se pri algoritmu po AHU za neurejena drevesa.

```
void top_down_tree_traversal_nodes_levels(tree T, list<list<node>> Lista) {
  queue<node> Q;
  int level=0;
  Q.append(T.root());
  do {
    list<node> nodes, childrennodes;
    do {
      node u=Q.pop();
      list<node> children;
      nodes.append(u);
      children=T.getnodechildren(u);
      for (int i=0; i<children.size(); i++) {
        childrennodes.append(children.get(i));
      }
    } while (!Q.empty());

    Lista.add(level++, nodes);
    for (int j=0; j<childrennodes.size(); j++) {
      Q.append(childrennodes.get(j));
    }
  } while (!Q.empty());
}
```

3.3.2 Algoritem po Valiente za neurejena drevesa

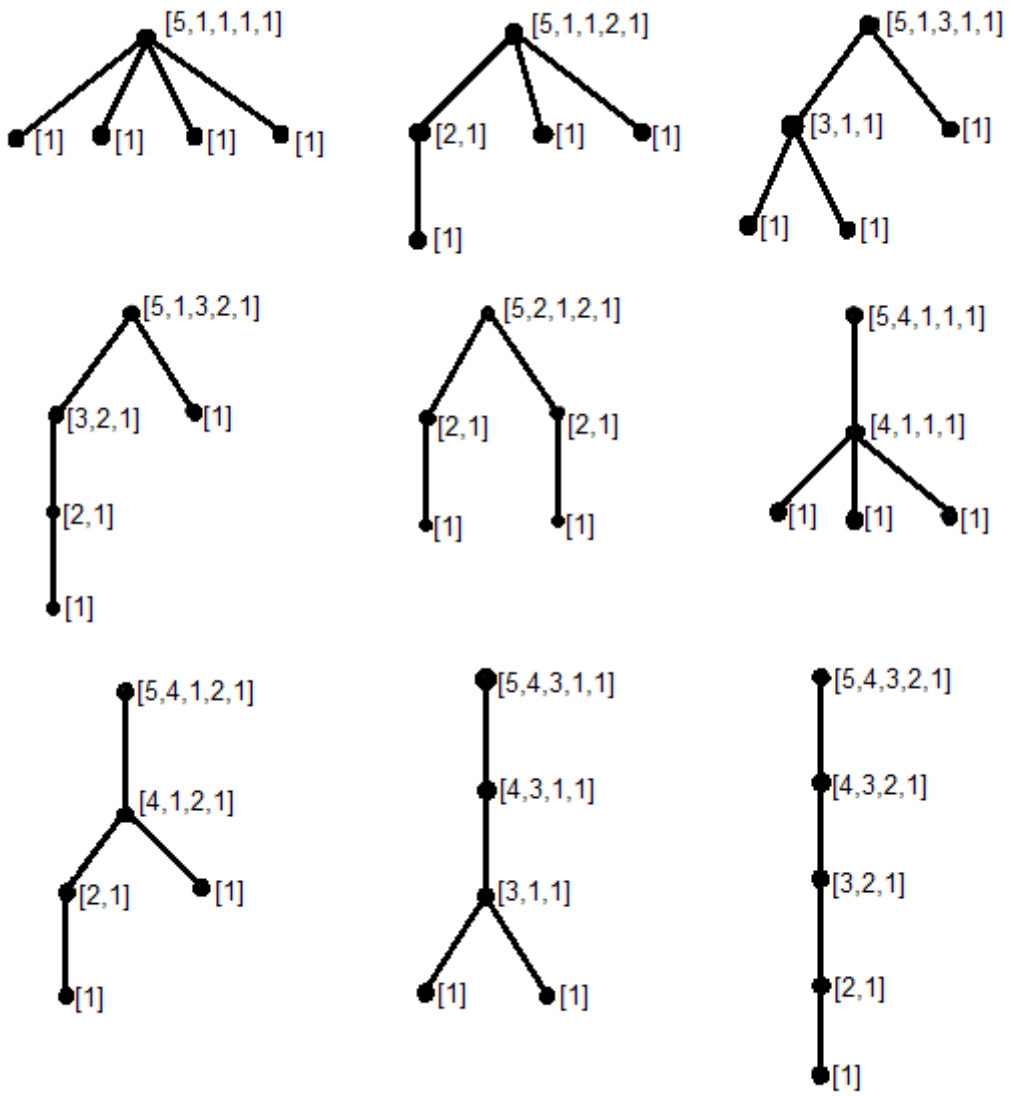


Slika 3.2: Izomorfna drevesa. Vozlišča so oštevilčena po vrstnem redu obiska med premim obhodom.

Neurejeni drevesi $T_1 = (V_1, E_1)$ in $T_2 = (V_2, E_2)$ iz slike 3.2 sta izomorfni. Bijekcija $M : V_1 \rightarrow V_2$, tako da $M = \{(u_1, v_1), (u_2, v_{10}), (u_3, v_{11}), (u_4, v_{12}), (u_5, v_{13}), (u_6, v_2), (u_7, v_3), (u_8, v_4), (u_9, v_5), (u_{10}, v_9), (u_{11}, v_6), (u_{12}, v_7), (u_{13}, v_8)\}$ predstavlja drevesni izomorfizem iz T_1 v T_2.

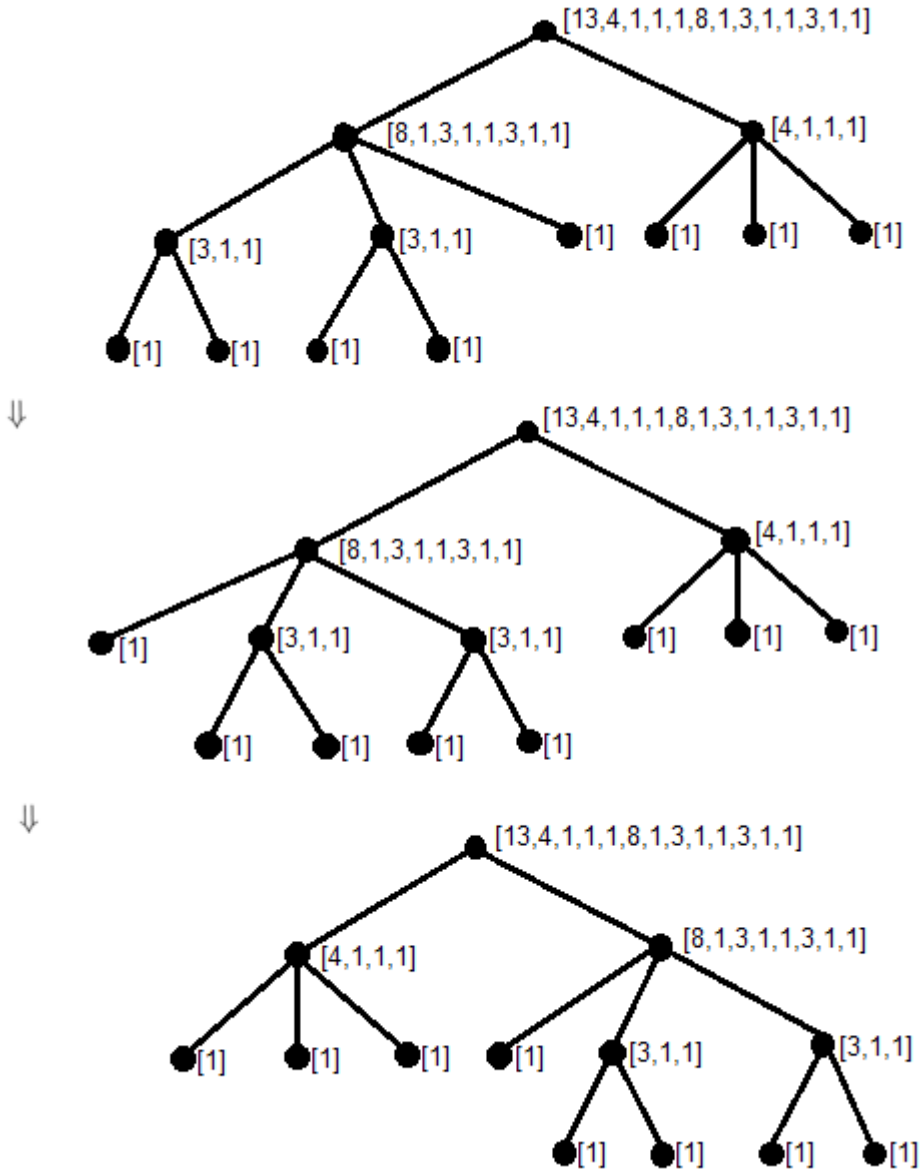
Definicija 3. Naj je $T = (V, E)$ neurejeno drevo z n vozlišči. Podpis (angl. isomorphism code) korena drevesa T predstavlja zaporedje n številk v rangu 1,...,n podana z

code[root[T]]=[size[root[T]],code[v_1],...,code[v_k]], kjer so vozlišča v_1,...,v_k otroci korena drevesa T razporejeni z nepadajočim leksikografskim urejanjem podpisov. Podpis neurejenega drevesa predstavlja pravzaprav podpis korena drevesa [1].



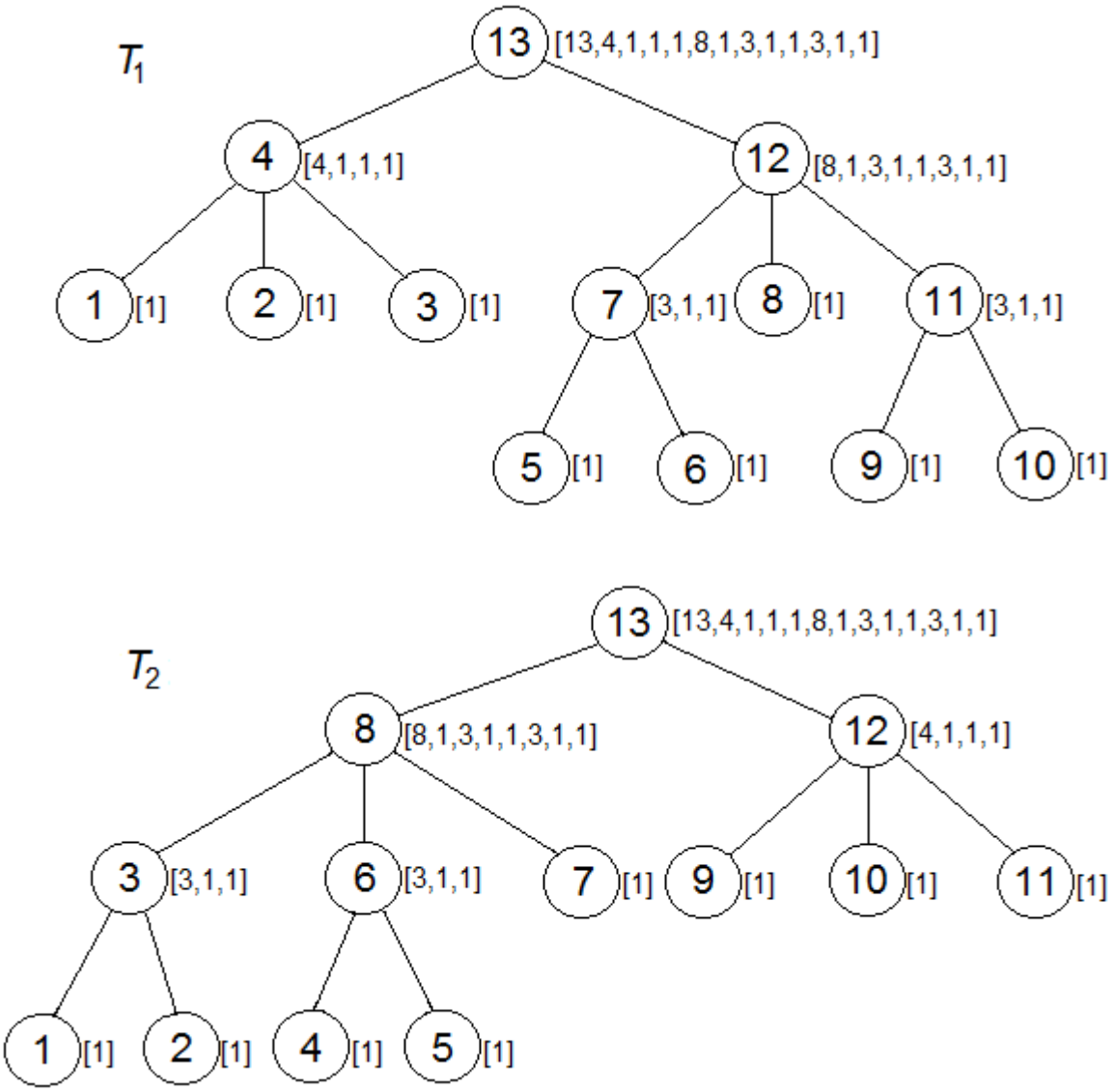
Slika 3.3: Podpisi devetih neizomorfnih neurejenih dreves. Podpis vsakega vozlišča je prikazan na desni strani vozlišča.

Podpis neurejenega drevesa lahko dobimo z izvedbo zaporednih permutacij otrok vozlišč v drevesu, s čimer bi preoblikovali neurejeno drevo v kanonično obliko urejenega drevesa, v katerem bi vrstni red sorojencev z leve proti desni odražal nepadajoči leksikografski vrstni red svojih podpisov, kot je prikazano na sliki 3.4. Dve neurejeni drevesi sta izomorfni, če sta njuni kanonično urejeni drevesi prav tako izomorfni.



Slika 3.4: Transformacija neurejenega drevesa v kanonično urejeno drevo. Podpis vsakega vozlišča je prikazan na desni strani vozlišča.

Alternativen preprost postopek za preizkušanje izomorfnosti dveh neurejenih dreves sestoji iz računanja podpisov za vsako vozlišče v drevesu med obratnim obhodom in šele nato sledi primerjanje podpisov korenov dreves.



Slika 3.5: Izvedba preprostega postopka za izomorfizem neurejenih dreves iz slike 3.2. Vozlišča so oštevilčena po vrstnem redu obiska med obratnim obhodom. Podpisi vozlišč so prikazani na njihovi desni strani.

Poglejmo izvedbo postopka za izomorfizem neurejenih dreves na sliki 3.5. Med obratnim obhodom drevesa, so podpisi za otroke vozlišč izračunani pred izračunom podpisov samih vozlišč. Na primer, podpis osmega vozlišča v obratnem obhodu drevesa T_2, *[8,1,3,1,1,3,1,1]*, dobimo z urejanjem v nepadajočem leksikografskem vrstnem redu podpisov tretjega, šestega in sedmega vozlišča, jih združimo in jim kot predpono dodamo še velikost poddrevesa T_2 ukoreninjeno v osmem vozlišču.

Testiranje izomorfnosti neurejenih dreves s tem algoritmom, ki temelji na podpisu vozlišč, velja le za neoznačena drevesa.

```
bool unordered_tree_isomorphic(tree T1, tree T2, nodearray<node> M) {
  if (T1.numberofnodes()≠T2.numberofnodes()) return false;

  nodearray<list<int>> code1(T1);
  nodearray<list<int>> code2(T2);
  assign_isomorphic_codes_to_all_nodes_of_T1_and_T2_in_postorder();

  if (code1[T1.root()]≡code2[T2.root()]) {
    build_tree_isomorphism_mapping();
    double_check_unordered_tree_isomorphism();
    return true;
  } else {
    return false;
  }
}
```

Podpisi vseh vozlišč iz obeh dreves T_1 in T_2 so izračunani z naslednjim postopkom z obratnim obhodom skozi drevesa. Čeprav bi se podpis vozlišča lahko izračunal z združevanjem podpisov otrok vozlišča, so podpisi vseh vozlišč še vedno potrebni, da bi se izračunala bijekcija $M : V_1 \rightarrow V_2$, ki ustreza neurejenemu izomorfizmu iz $T_1 = (V_1, E_1)$ v $T_2 = (V_2, E_2)$.

```
void assign_isomorphic_codes_to_all_nodes_of_T1_and_T2_in_postorder(){
  isomorphic_codes_nodes(T1, code1);
  isomorphic_codes_nodes(T2, code2);
}
```

```
void isomorphic_codes_nodes(tree T, nodearray<list<int>> isomorphiccode) {
  node u, v;
  array<int> A;
  list<array<int>> L;
  int code;
  list<node> Lnodes;

  postorder_tree_list_traversal(T, Lnodes);
  forall(u, Lnodes) {
    if (T.isleaf(u)) {
      isomorphiccode[u].append(1);
    } else {
      L.clear();
      code=1;
      forallchildren(v, u) {
        code+=isomorphiccode[v].head();
        list_to_array(isomorphiccode[v],A);
        L.append(A);
      }
      sort(L); //RadixSort or other sort method can be used
      isomorphiccode[u].append(code);
      forall(A, L) {
        forall(code, A) {
          isomorphiccode[u].append(code);
        }
      }
    }
  }
}
```

V kolikor sta drevesi $T_1 = (V_1, E_1)$ in $T_2 = (V_2, E_2)$ izomorfni, se lahko ustrezna bijekcija $M : V_1 \rightarrow V_2$ izračuna na podlagi podpisov vozlišč od T_1 v T_2. Vozlišče $u \in V_1$ se preslika v vozlišče $v \in V_2$, oziroma $(u, v) \in M$, če in samo če code1[u]=code2[v] in (parent[u], parent[v]) $\in M$.

```
void build_tree_isomorphism_mapping() {
  node u=T1.root();
  node v=T2.root();
  M[u]=v;
  nodearray<bool> mappedto(T2, false);
  list<node> L1;
  preorder_tree_list_traversal(T1, L1);
  L1.pop();  //node u is already mapped
  forall(u, L1) {
    forallchildren(v, M[T.parent(u)]) {
      if (code1[u]≡code2[v] AND !mappedto[v] AND (T2.isroot(v) OR
M[T1.parent(u)]≡T2.parent(v))) {
        M[u]=v;
        mappedto[v]=true;
        break;
      }
    }
  }
}
```

Preverjanje neurejenega drevesnega izomorfizma, ki sledi, čeprav je odveč, zagotavlja, da je algoritem pravilno implementiran. Preverja, da je *M* neurejen drevesni izomorfizem iz T_1 v T_2 po definiciji 2.

```
void double_check_unordered_tree_isomorphism() {
  node u;
  forallnodes(u, T1) {
    if (M[u]≡null OR T1[u]≠T2[M[u]])
      error_handler ("Wrong implementation of unordered tree isomorphism");
    if (!T1.isroot(u) AND (T2.isroot(M[u]) OR
(M[T1.parent(u)]≠T2.parent(M[u]))))
      error_handler ("Wrong implementation of unordered tree isomorphism");
  }
}
```

Uporabljene so bile tudi naslednje pomožne funkcije, s katerimi se podpisi pretvorijo v niz celih števil, preden se urejanje izvede, in metode za primerjavo podpisov.

```
void list_to_array(list L, array A) {
  A.resize(1, L.length());
  int i=1;
  forall(element, L) {
    A[i++]=element;
  }
}
```

```
int compare(list L1, list L2) {
  listelement p=L1.first();
  listelement q=L2.first();
  while (p≠null AND q≠null) {
    if (L1[p]<L2[q]) return -1;
    if (L1[p]>L2[q]) return 1;
    p=L1.succ(p);
    q=L2.succ(q);
  }
  if (q≠null) return -1; //p==null AND q!=null
  if (p≠null) return 1; //p!=null AND q==null
  return 0; //p==null AND q==null
}

bool operator≡(list<int> L1, list<int>L2) {
  return compare(L1, L2)≡0;
}
```

Izrek 1. Algoritem za neurejen drevesni izomorfizem se izvaja časovno zahtevnostjo $O(n^2)$ in s prostorsko zahtevnostjo $O(n)$, kjer je n število vozlišč v drevesu.

Dokaz: Naj bosta $T_1 = (V_1, E_1)$ in $T_2 = (V_2, E_2)$ dve neurejeni drevesi z n vozlišči. Čas potreben za urejanje podpisov otrok vozlišča v leksikografskem vrstnem redu je sorazmeren številu otrok vozlišča in dolžini podpisov, ki jih je treba urediti. Čas je odvisen tudi od načina izvajanja urejanja. Če bi za urejanje uporabili korensko urejanje, bi bil potreben čas linearen s skupno dolžino nizov tudi v primeru, če so nizi različno dolgi. Časovna zahtevnost urejanja podpisov je vsota časovne zahtevnosti vseh vozlišč drevesa.

Če je eno vozlišče v drevesu, je časovna zahtevnost $T(1)=1$. Če sta dve vozlišči v drevesu je $T(2)=T(1)+2$. V splošnem primeru, ko je n vozlišč v drevesu je časovna zahtevnost podana z rekurzivno formulo $T(n)=T(n-1)+n$. Zapišimo nekaj členov te formule.

$$T(n)=T(n-1)+n \tag{3.1}$$

$$T(n-1)=T(n-2)+n-1 \tag{3.2}$$

$$T(n-2)=T(n-3)+n-2 \tag{3.3}$$

Če v enačbo 3.1 vstavimo za $T(n-1)$ enačbo 3.2 dobimo enačbo 3.4.

$$T(n)=T(n-2)+n-1+n \tag{3.4}$$

Če v istem smislu nadaljujemo in vstavimo enačbo 3.3 v enačbo 3.4 dobimo enačbo 3.5.

$$T(n)=T(n-3)+n-2+n-1+n \tag{3.5}$$

Iz zgornjih enačb je razvidno, da velja naslednja enačba:

$$T(n)=T(n-k)+k*n-k(k-1)/2 \tag{3.6}$$

kjer je člen $k(k-1)/2$ seštevek prvih $k-1$ naravnih števil.

Če v enačbo 3.6 postavimo substitucijo $n-k=1$, dobimo enačbo 3.7.

$$T(n)=T(1)+(n-1)*n-(n-1)*(n-2)/2 \tag{3.7}$$

Iz 3.7 ugotovimo, da je časovna zahtevnost algoritma $T(n)=O(n^2)$.

3.3.3 Algoritem po Aho, Hopcroft in Ullman

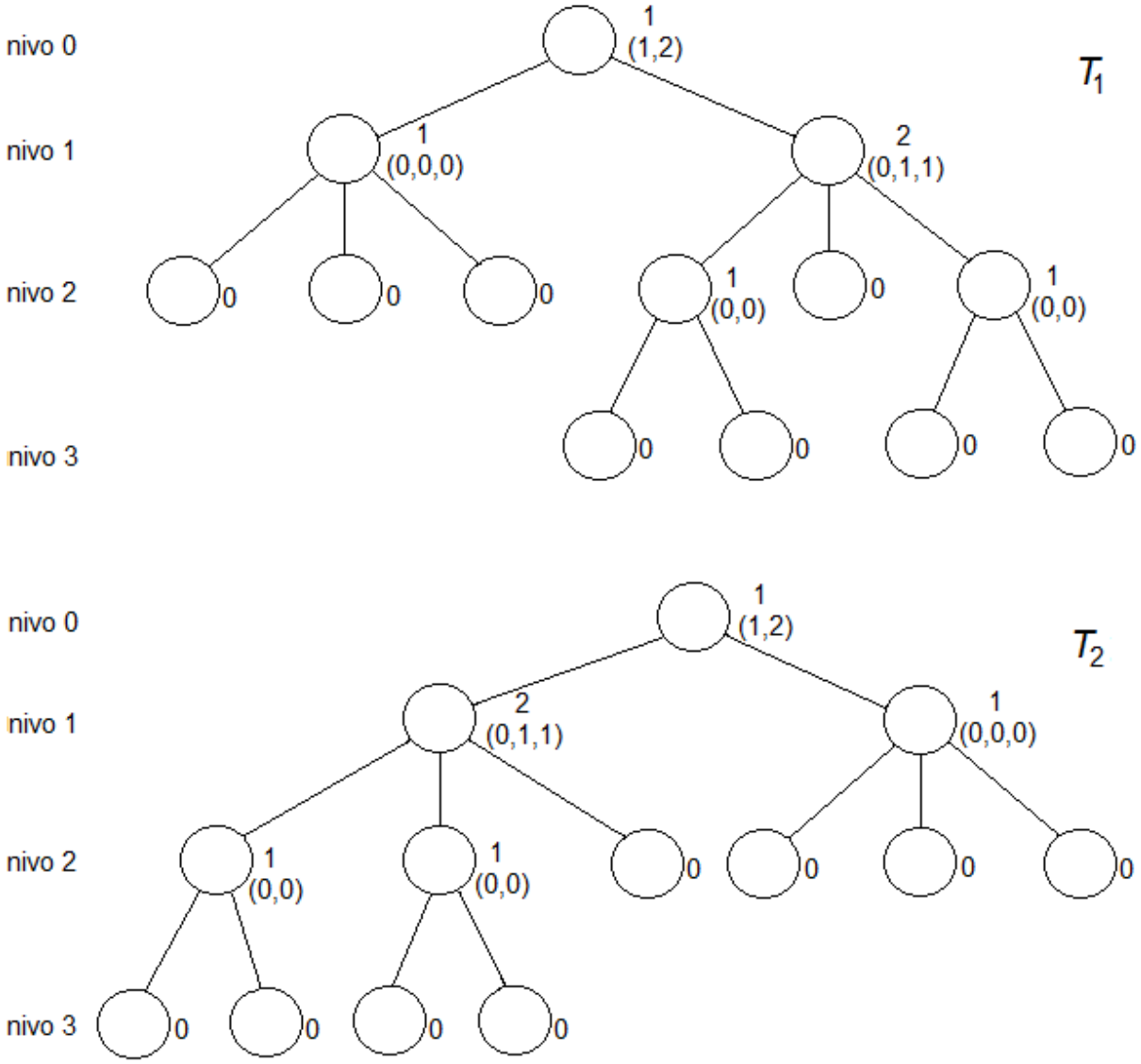
Dve ukoreninjeni drevesi sta izomorfni, če lahko preslikamo eno drevo v drugo s permutiranjem vrstnega reda otrok vozlišč.

Naslednji algoritem [4], napisan s funkcijo *unordered_tree_isomorphic(T1, T2)* ima linearno časovno zahtevnost sorazmerno s številom vozlišč. Algoritem dodeli cela števila vozliščem obeh dreves, začenši pri vozliščih na nivoju (angl. level) *k* nato pa navzgor proti korenoma. Drevesi sta izomorfni, če in samo če je dodeljena enaka številka njunima korenoma.

Testiranje izomorfnosti neurejenih dreves s tem algoritmom velja za neoznačena drevesa.

```
bool unordered_tree_isomorphic(tree T1, tree T2) {
  node u;
  list<node> L1, L2;
  list<list<node>> Lista1, Lista2;
  array<int> NodesTuples1[1, T1.numberofnodes()];
  array<int> NodesTuples2[1, T2.numberofnodes()];

  //group nodes of T1 and T2 having same depth into separate level group
  top_down_tree_traversal_nodes_levels(T1, Lista1);
  top_down_tree_traversal_nodes_levels(T2, Lista2);

  //assign the integer 0 to all leaves of T1 and T2
  forallnodes(u, T1) {
    if (T1.isleaf(u)) NodesTuples1[u]=0;
  }
  forallnodes(u, T2) {
    if (T2.isleaf(u)) NodesTuples2[u]=0;
  }

  //generate tuples for vertices that are on the same level
  //starting from the 'bottom of the tree', i.e. level k
  //those are the nodes having greatest depth
  int level=Lista1.size()-1;
  array<array<int>> S1, S1´, S2, S2´;

  //L1 (L2) is the list of all nodes of T1 (T2) at 'level 0'
  L1=Lista1.get(level);
  L2=Lista2.get(level);
  do {
    forallnodes(u, L1) {
      //integer assigned to node u is the next component of the tuple
assigned to the parent of u
      NodesTuples1[u.getparent()].add(NodesTuples1[u]);
    }
```

```
    //S1 is the sequence of tuples created for the nodes of T1 on level i
    forallnodestuplesonupperlevel(v, Lista1.get(level-1)) {
      S1.add(NodesTuples1[v]);
    }
    //S1´ is actually lexicographic sorted S1
    S1´=sort(S1); //RadixSort or other sort method can be used
    forallnodes(u, L2) {
      NodesTuples2[u.getparent()].add(NodesTuples2[u]);
    }
    forallnodestuplesonupperlevel(v, Lista2.get(level-1)) {
      S2.add(NodesTuples2[v]);
    }
    S2´=sort(S2); //RadixSort or other sort method can be used
    if (S1´ ≠ S2´) return false; //T1 and T2 are not isomorphic

    level--;
    L1.clear(); //L1 is empty list of nodes
    for (int k=1; k≤number of distinct tuples of S1´; k++) {
      forallnodes(u, T1, on level represented by k-th tuple of S1´) {
        NodesTuples1[u]=k; //assign the integer k to node u
        L1.append(u);      //append node u to L1
      }
    }
    forallleaves(u, T1, on level i) {
      L1.add(u); //append all leaves of T1 on level i at the front of L1
    }

    L2.clear();
    for (int k=1; k≤number of distinct tuples of S2´; k++) {
      forallnodes(u, T2, on level represented by k-th tuple of S2´) {
        NodesTuples2[u]=k;
        L2.append(u);
      }
    }
    forallleaves(u, T2, on level i) {
      L2.add(u);
    }
  } while (level>0); //while not reached 'level 0'

  if (NodesTuples1[0]=NodesTuples2[0]) {
    //roots of T1 and T2 are assigned same integer => they are isomorphic
    return true;
  } else {
    return false;
  }
}
```



Slika 3.6: Številke, dodeljene vozliščem po algoritmu za drevesni izomorfizem.

Izrek 2. Preverjanje enakovrednosti dreves po tem algoritmu ima časovno zahtevnost *O(n)*.

Dokaz: Dokaz sledi iz postopka za leksikografsko urejanje nizov znakov z različno dolžino. Naj sta $T_1 = (V_1, E_1)$ in $T_2 = (V_2, E_2)$ dve neurejeni drevesi z n vozlišči. Čas, ki se porabi za dodeljevanje celih števil vozliščem, ki niso listje na nivoju i, je sorazmeren številom vozlišč na nivoju i-1. S seštevanjem čez vse nivoje dobimo časovno zahtevnost *O(n)*. Tudi dodeljevanje števil listom je sorazmerno številom vozlišč n, zato ima algoritem časovno zahtevnost *O(n)* [2].

Poglavje 4 Eksperimentalna primerjava algoritmov

4.1 Generiranje naključnih dreves

Za potrebe eksperimentalnega ovrednotenja algoritmov v programskem jeziku Java smo razvili programski modul, ki generira poljubno število dreves z določenim številom vozlišč.

Modul naključno generira in dodaja vozlišča. Kot vhodni parameter prejme celo število, ki je število vozlišč v drevesu. Dodaten programski del generira izomorfne dvojnike obstoječih dreves. Poda se število dvojnikov, ki se generirajo iz osnovnega drevesa s permutiranjem naključnih povezav med vozlišči. Primerjali smo ali je osnovno drevo izomorfno z vsemi njegovimi dvojniki z uporabo algoritmov.

Implementirali smo tudi ustrezne in potrebne razrede (Tree, Node, Isomorphism) ter pomožne funkcije.

V sklopu diplomske naloge smo generirali različne skupine neoznačenih dreves, nad katerimi smo izvedli programsko implementacijo algoritmov za določanje izomorfnosti korenskih dreves. Začetnemu, naključno generiranjemu drevesu, smo permutirali naključne povezave med vozlišči in tako dobili druga drevesa, izomorfna z njim. Postopek smo ponovili nad drevesi z različnim številom vozlišč:

- petim drevesom (prvo drevo je imelo 100 vozlišč, drugo 300, tretje 500, četrto 700 in peto 900 vozlišč) smo permutirali naključne povezave in tako dobili za vsako drevo po 40 izomorfnih dreves;

- petim drevesom (prvo drevo s 1000 vozlišči, drugo 3000, tretje 5000, četrto 7000 in peto 9000 vozlišč) smo permutirali naključne povezave in tako dobili za vsako drevo po 20 izomorfnih dreves;

- petim drevesom (prvo drevo je imelo 10000 vozlišč, drugo 30000, tretje 50000, četrto 70000 in peto 90000 vozlišč) smo permutirali naključne povezave in tako dobili za vsako drevo po 10 izomorfnih dreves;

- enemu drevesu s 100000 vozlišč, smo permutirali naključne povezave in tako dobili 3 izomorfna drevesa;

- dvema drevesoma (prvo drevo je imelo 200000 vozlišč in drugo 300000 vozlišč) smo permutirali naključne povezave in tako dobili za vsako drevo po 1 izomorfno drevo.

Dobljeni rezultati so prikazani v poglavju 4.3.

4.2 Testno okolje

Razvojno ter testno okolje, ki smo ga uporabili pri delu je bilo:

prenosni računalnik Lenovo W510

Intel Core i7 CPU Q820 @ 1,73GHz

8GB RAM

Trdi disk 500GB

64-bitna različica operacijskega sistema Windows 7 Enterprise SP1

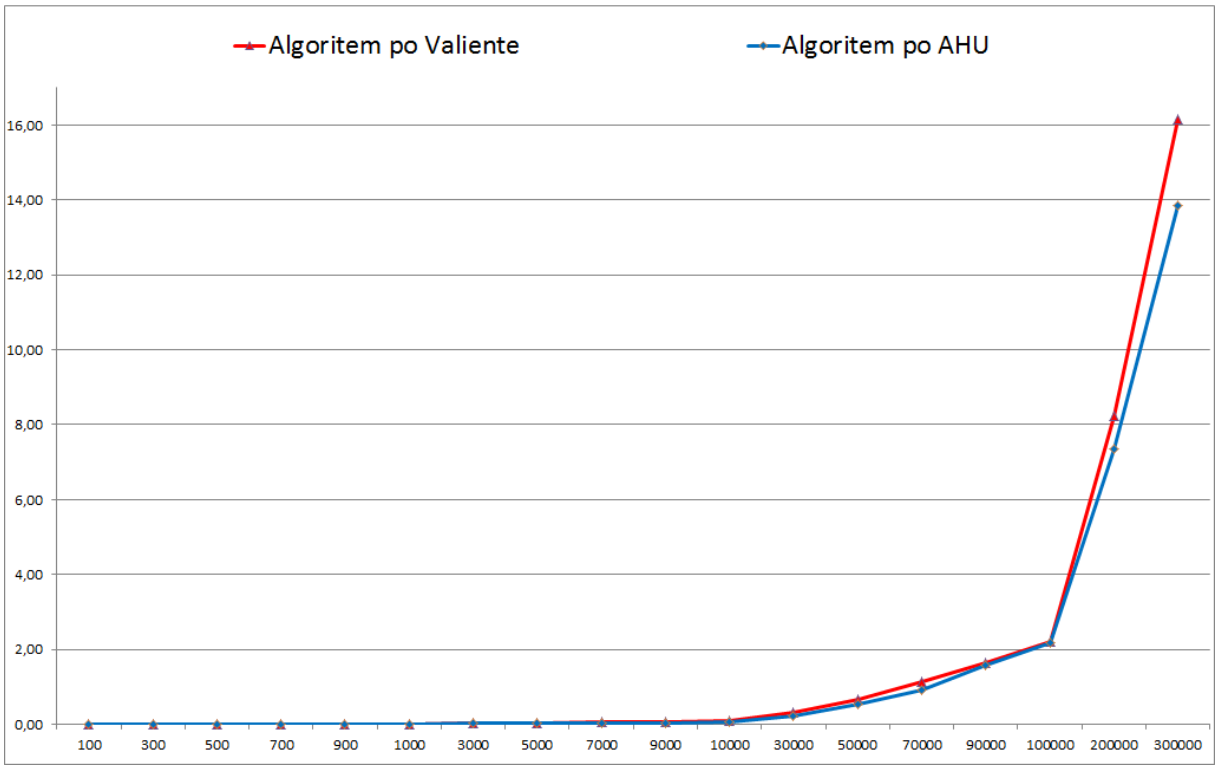
NetBeans IDE 7.3 z Java JDK 1.7.0_15

4.3 Rezultati

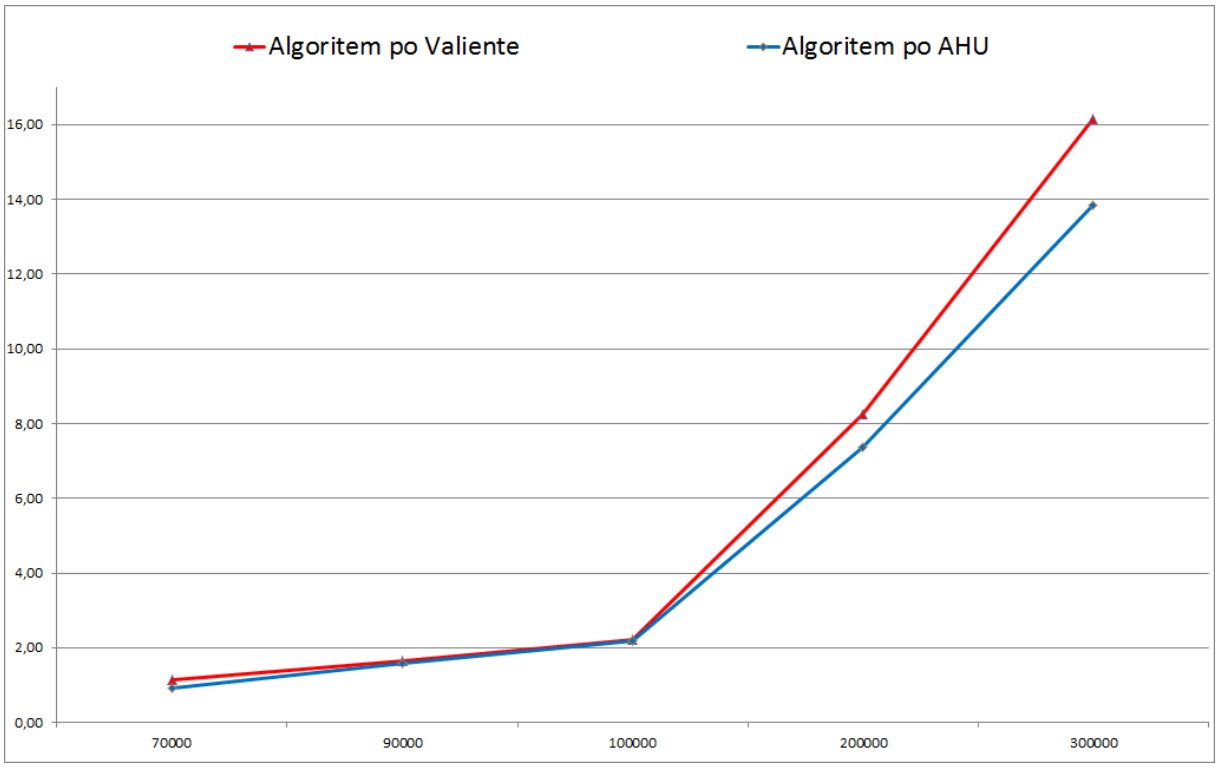
Z implementacijo omenjenih algoritmov za izomorfnost neurejenih dreves (po Valiente in po AHU), smo dobili naslednje rezultate:

število vozlišč v drevesu	število izomorfnih dreves	skupni čas izvajanja v sekundah - algoritem po Valiente	povprečni čas / poskus v sekundah - algoritem po Valiente	skupni čas izvajanja v sekundah - algoritem po AHU	povprečni čas / poskus v sekundah - algoritem po AHU
100	40	0,1658	0,0041	0,1026	0,0026
300	40	0,2305	0,0058	0,1399	0,0035
500	40	0,2707	0,0068	0,1553	0,0039
700	40	0,3476	0,0087	0,1647	0,0041
900	40	0,3697	0,0092	0,1948	0,0049
1000	20	0,3365	0,0168	0,1643	0,0082
3000	20	0,6349	0,0317	0,3487	0,0174
5000	20	0,8137	0,0407	0,4555	0,0228
7000	20	1,1257	0,0563	0,4676	0,0234
9000	20	1,3637	0,0682	0,6227	0,0311
10000	10	0,9647	0,0965	0,5661	0,0566
30000	10	3,0872	0,3087	2,0795	0,2080
50000	10	6,5420	0,6542	5,3746	0,5375
70000	10	11,4385	1,1439	9,3005	0,9300
90000	10	16,3714	1,6371	15,8498	1,5850
100000	3	6,6424	2,2141	6,5739	2,1913
200000	1	8,2501	8,2501	7,3565	7,3565
300000	1	16,1449	16,1449	13,8514	13,8514

Tabela 4.1: Čas izvajanja algoritmov (Valiente, AHU).



Graf 4.1: Čas izvajanja algoritmov (Valiente, AHU).



Graf 4.2: Čas izvajanja algoritmov (Valiente, AHU).

Na grafu 4.1 sta prikazana časa izvajanja algoritmov: z rdečo barvo je čas izvajanja algoritma po Valiente in z modro algoritma po AHU. Na abscisni osi je število vozlišč dreves, medtem ko na ordinatni je povprečni čas enega poskusa za preverjanje izomorfnosti neurejenih dreves.

Graf 4.2 je enak z grafom 4.1 za vsa drevesa s številom vozlišč večje ali enako 70000. V grafu 4.2 nismo prikazali časov izvajanja algoritma za drevesa, ki so imela števila vozlišč manjša od 70000, ker so ti časi nepomembni, saj so krajši od ene sekunde.

Abscisna os x na grafih 4.1 in 4.2 je prikazana logaritemsko, zato grafi eksponentno naraščajo.

Na podlagi izmerjenih časov, podanih v tabeli 4.1 ter grafih 4.1 in 4.2, lahko sklepamo, da je algoritem po AHU malenkost hitrejši od algoritma po Valiente. Iz naših testnih rezultatov ugotavljamo, da je 100000 vozlišč v drevesih vrednost, pri kateri preverjanje drevesnega izomorfizma po obeh algoritmih traja skoraj enako dolgo.

Če upoštevamo še to, da smo pri obeh algoritmih za urejanje uporabili isto po meri napisano javansko metodo *Java.util.Arrays.sort()*, lahko zaključimo, da so časi izvajanja algoritmov odvisni le od implementacije algoritma.

Poglavje 5 Sklepne ugotovitve

V diplomski nalogi smo implementirali in nato primerjali nekaj algoritmov za preverjanje izomorfnosti dreves, s poudarkom na algoritmih za določanje izomorfizma neurejenih dreves.

Iz dobljenih rezultatov sklepamo, da oba algoritma (po Valiente in po AHU) potrebujeta skoraj enak čas za preverjanje enakosti, oziroma izomorfnosti dreves, če imajo drevesa po 100000 vozlišč. V drugih primerih se algoritem po AHU izvaja hitreje kot algoritem po Valiente.

Literatura

[1] G. Valiente, *Algorithms on Trees and Graphs*, Springer, 2010.

[2] Aho, Hopcroft in Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[3] T. Sager, A. Bernstein, M. Pinzger in C. Kiefer, "Detecting Similar Java Classes Using Tree Algorithms", v zborniku *Proceedings of the 2006 international workshop on Mining software repositories*, Shanghai, China, maj 2006, str. 65-71.

[4] G. Valiente. *Tree Isomorphism and Related Problems* [Online]. Dosegljivo: http://www.cs.upc.edu/~valiente/graph-00-01-c.pdf.

[5] R. Škrekovski. (2010). *Diskretne strukture II, zapiski predavanj* [Online]. Dosegljivo: http://www.fmf.uni-lj.si/~skreko/Gradiva/DS2-skripta.pdf.

[6] J. Mihelič. *Algoritmi in podatkovne strukture 1* [Online]. Dosegljivo: https://ucilnica.fri.uni-lj.si/pluginfile.php/25979/mod_resource/content/0/P12-Drevesa.pdf.

[7] I. Kononenko, *Načrtovanje podatkovnih struktur in algoritmov*, Fakulteta za računalništvo in informatiko, 1996.